Auddard & Co.

Arithmetic Expression Evaluator in C++
Software Architecture Document

Version 1.2

Revision History

Date	Version	Description	Author
09/Nov./23	1.0	Filled out the Introduction section and Architectural Representation. Updated TOC	Audrey Pan, Sriya Annem, Morgan Nguyen, Lauren D'Souza, Ella Nguyen, Tanu Sakaray
10/Nov./23	1.1	Filled the Architectural goals and Constraints section and the Logical View section	Audrey Pan, Sriya Annem, Morgan Nguyen, Lauren D'Souza, Ella Nguyen, Tanu Sakaray
12/Nov./23	1.2	Finished the interface description section and the quality section.	Audrey Pan, Sriya Annem, Morgan Nguyen, Lauren D'Souza, Ella Nguyen, Tanu Sakaray

Table of Contents

Software Architecture Document

1. Introduction

1.1 Purpose

The purpose of this Software Development Plan is to provide a comprehensive strategy and framework for the development of the Arithmetic Expression Evaluator in C++ project. This plan outlines the scope, objectives, roles, responsibilities, and deliverables for the project, ensuring that it results in a functional software feature.

The following people contributed and plan to use this *Software Development Plan*:

- **Project Manager** Audrey Pan uses it to plan the project schedule and resource needs, and to track progress against the schedule.
- **Project Team Members** Tanushri Sakaray, Sriya Annem, Ella Nguyen, Lauren D'Souza, Morgan Nguyen use it to understand what they need to do, when they need to do it, and what other activities they are dependent upon.

1.2 Scope

- This *Software Development Plan* describes the overall plan for developing the Arithmetic Expression Evaluator to be used by the Auddard & Co project, including deployment of the feature.
- This software will be able to handle expressions enclosed within parentheses to determine the correct order of evaluation. The scope also includes the creation of various project management artifacts, including a project plan, a requirements document, a design document, and test cases.
- Additionally, the project entails the development of a user-friendly command-line interface for entering expressions and viewing calculated results.
- Robust error handling mechanisms will be implemented to manage scenarios such as division by zero or invalid expressions.
- Initially, the project assumes input values to be integers. However, the scope allows for potential future changes, such as accommodating floating-point input values, which may be introduced through change requests during project execution.

1.3 Definitions, Acronyms, and Abbreviations

See the Project Glossary in section 6.1.

1.4 References

SN	Title	Published Date	Publishing Organization
1.	EECS348: Term Project in C++ Specification	09/Sep/23	Professor Hossein Saiedian
2.	Expression Parser		Lawrence University
3.	The rules for the order of operations: The case of an inservice teacher	02/Mar/16	HAL Open Science

1.5 Overview

The *Software Development Plan* (SDP) for the Arithmetic Expression Evaluator project in C++ encompasses a clear project overview, including its purpose, scope, objectives, and deliverables. It outlines the project's organizational structure, external interfaces, and states the roles and responsibilities of team members and stakeholders. In the management process section, it details cost estimates, schedules, key phases, and milestones, along with the methods for ongoing project monitoring. The annexes provide insights into the chosen software development process, including methodologies, tools, and techniques, while the appendix contains a Project Glossary for clarifying essential project-related terminology, which serves as a comprehensive guide for the project's successful execution.

2. Architectural Representation (Sriya)

The software architecture for the Arithmetic Expression Evaluator system in C++ encompasses various architectural views to represent its structural and behavioral aspects. The architectural representation adopts a multi-view approach to provide a comprehensive understanding of the system. The necessary views include the Logical View, the Process View, the Development View, and the Physical View.

- **The Logical View** primarily contains the design elements focusing on the internal structure of the system. This encompasses components such as expression parsing, operator precedence definition, and arithmetic evaluation logic.
- **The Process View** illustrates the system's dynamic behavior, detailing the interactions and communication among different components during expression evaluation. It delineates how the program handles input, tokenization, and the evaluation process itself.
- **The Development View** highlights the structure to support software development, including the programming environment, tools, and dependencies required for the project.
- **The Physical View** emphasizes the deployment and distribution aspects, defining how the system is mapped onto physical hardware. It covers aspects such as memory utilization, performance considerations, and potential optimizations for running the system efficiently on different computers.

Each view is crucial in offering a comprehensive representation of the software architecture, encompassing elements crucial for understanding the system's structure and behavior.

3. Architectural Goals and Constraints

Software Requirements & Objectives

- **Expression Parsing and Evaluation:** The primary objective is to develop a robust C++ program capable of parsing and evaluating arithmetic expressions containing operators such as +, -, *, /, %, and ^, along with numeric constants. The program should adhere to the order of operations (PEMDAS) and support expressions with parentheses for precedence and grouping.
- **Development Process Emphasis:** This project emphasizes having a comprehensive development process that goes beyond the final product. Deliverables should contain a well-crafted project plan, requirements document, design document, a set of rigorous test cases, and the fully realized product. The development process should maintain coherence and alignment between project stages.

Special Constraints

- **Documentation Requirements:** Deliverables must include a project management plan, requirements document, design document, and test cases. These documents should be well-documented, coherent, and interlinked to ensure clarity and consistency when executing.
- **Tokenization and Data Structure:** Implement a function for tokenizing input expressions and design a suitable data structure to represent the expression’s structure.
- **Operator Precedence:** Define operator precedence based on PEMDAS rules and implement logic to evaluate expressions while considering operator precedence.
- **Numeric Constants:** Develop the capability to recognize and calculate numeric constants in the input expressions, initially assuming integers and potentially accommodating floating-point values in the future.

- **User Interface:** Create a user-friendly and legible command-line interface allowing users to input expressions and view calculated results.
- **Error Handling:** Implement robust error handling to manage scenarios such as division by zero or invalid expressions.
- **Object-Oriented Programming:** Utilize object-oriented programming principles to structure the code effectively.
- **Unit Testing:** Develop unit tests to verify the correctness of the expression evaluator.
- **User Manual/README:** Provide a user manual or README file explaining how to use the program, including examples.

4. Logical View

4.1 Overview

The decomposition of the design model in terms of its package hierarchy and layers are as follows. Brief descriptions and breakdowns of the classes are provided in the next section (4.2)

- Expression Evaluation Package
- Expression Parsing Package
- Operator Handler Package
- Parenthesis Handler Package
- Constants Recognition Package
- User Interface (UI) Package
- Error Handling Package
- Main Application Package

4.2 Architecturally Significant Design Modules or Packages

- Expression Evaluation Package
    - Description: The main package that handles expression evaluation
    - Classes:
        - ‘Expression Evaluator’: Manages the overall expression evaluation process
        - ‘Token’: Represents the tokenized elements in the expression that the user inputs
        - ‘Expression Tree’: The hierarchical structure of the expression
- Expression Parsing Package
    - Description: Creates expression tree and handles tokenization
    - Classes:
        - ‘Expression Parser’: Implements functions to tokenize the expression that the user inputs
        - ‘Parser Utilities’: Provides utility functions for parsing the expression
- Operator Handler Package
    - Description: Handles and manages operator precedence and evaluation logic of the expression
    - Classes:
        - ‘Operator Precedence’: Defines the precedence of the operators in the expression following the PEMDAS order of operations
        - ‘Expression Evaluator’: Implements logic to evaluate the expression inputted by the user based on the operator precedence defined previously
- Parenthesis Handler Package
    - Description: Manages identifying and evaluating the expressions that are inside the parenthesis
    - Classes:
        - ‘ParenthesisHandler’: Implements mechanisms to identify and evaluate expressions that have parenthesis
- Constants Recognition Package
    - Description: Recognizes and handles numeric constants in the inputted expression from

the user
    - Classes:
        - ‘Constant Recognizer’: Implements functions that identify and process numeric constants in the expression
- User Interface (UI) Package
    - Description: Handles interactions and takes input expressions from the users and displays results
    - Classes:
        - ‘User Interface’: Manages the command-line interface for the user to input their expression and the output/results
- Error Handling Package
    - Description: Implements robust error handling for situations like dividing by zero, invalid expressions, or other errors that might occur
    - Classes:
        - ‘Error Handler’: Manages error situations and provides descriptive and informative error messages
- Main Application Package
    - Description: Integrates all components of the application to create a fully functional arithmetic expression evaluator according to the requirements
    - Classes:
        - ‘Main Application’: Coordinate the interaction between the different modules and functions to achieve overall functionality and desired results

5. Interface Description

User Interface (Command-Line Interface)

Screen Format:

- The command-line interface should have clear prompt for users to input arithmetic expressions
- A well-formatted display of the calculated results.

Valid Inputs:

- Accept arithmetic expressions using operators +, -, *, /, %, and ^
- Numeric constants (initially integers with potential support for floating-point values in the future.)
- Parentheses for grouping

Resulting Outputs:

- Display the evaluated result of the entered expression
- provide clear feedback in case of invalid expressions

User Interaction:

- Prompts users with messages such as “Enter an arithmetic expression.”
- After receiving input, the program will process it and display the result or an error message

Ex.

Enter an arithmetic expression: (3 + 5) * 2
Result: 16

Error Handling Interface

Screen Format

- Clear and informative error messages

Valid Inputs

- Any input adhering to the defined arithmetic expression syntax

Resulting Outputs:

- Display relevant error messages for specific scenarios (e.g., division by zero, invalid expressions).
- Prompt users to correct their inputs

User Interaction:

- Provide guidance on how to correct errors
- If the expression is invalid, prompt users to enter a new one

Ex.

Error: Division by zero. Please enter a valid expression.
Enter a corrected expression: 5/2
Result: 2.5

Toeknization Interface:

Screen Format:

- Internal processing that is not directly visible to the user

Valid Inputs:

- Arithmetic expressions

Resulting Outputs:

- Tokenized representation of the input expression

User Interaction:

- No direct user interaction; part of the internal parsing process

Ex.

```
// internal representation of tokenized expression
[ "(", "3", "+", "5", ")", "*", "2"] //representation for ( 3 + 5 ) * 2
```

6. Quality

[A description of how the software architecture contributes to all capabilities (other than functionality) of the system: extensibility, reliability, portability, and so on. If these characteristics have special significance, such as safety, security or privacy implications, they must be clearly delineated.]

- **Extensibility** - The modular design of this arithmetic calculator aids in its extensibility, as new modules can be added in the future to adapt to changing user requirements.
- **Reliability** - The robust error handling mechanisms implemented contribute to the system's reliability. For instance, thorough validation of input expressions, detection of division by zero, and handling of invalid expressions enhance the overall reliability of the Arithmetic Expression Evaluator.
- **Portability** - The Development View addresses portability concerns by defining a clear structure that separates platform-dependent and platform-independent components. The design ensures that the core logic of expression parsing and evaluation remains independent of the underlying hardware or operating system.
- **Performance** - Performance considerations are found in the Physical View, and involve efficient memory utilization. The architecture allows for efficient deployment, aiming to minimize resource consumption during expression evaluation. The modular design of the system also supports

scalability, ensuring that performance remains satisfactory even as the complexity of expressions or the volume of concurrent users increases.

- **Security** - While the current scope of the project may not involve sensitive data, the architecture is designed with security and privacy in mind. Input validation mechanisms prevent potential security vulnerabilities, and the modular structure facilitates the implementation of additional security measures if required in the future.
- **Usability** - The user-friendly command-line interface enhances the system's usability, allowing easy expression input and result interpretation. Clear error messages improve the overall user experience.
- **Maintainability -** The architecture prioritizes maintainability with a clear separation of concerns and well-documented components. This facilitates ease of maintenance for future updates or enhancements.